

Scientific Application Performance

Serial and Parallel Performance
Monitoring and Tuning

Agenda

- 9:30-10:45 • Tools and Single CPU Performance
- 10:45-11:00 • Break
- 11:00-12:00 • Parallel Performance
- 12:00-1:00 • Lunch
- 1:00-3:00 • Play Time

Intent

- A starting point for performance analysis tools
- Once CPU hotspots are identified
 - Understand why they perform poorly
 - First steps in increasing performance
- Parallel programming performance issues

Outline

- Brief overview of common tools
 - Timers, Gprof
 - Hardware Counters, Valgrind
 - Jumpshot/FPMPI
- Common performance bottlenecks
 - General single CPU
 - Cache performance
- Parallel performance issues

Performance Measurement

- *Where* is time spent in application
 - Routine timers, Communication patterns
- *Why* is time being spent there
 - Single CPU performance
 - Operations
 - Cache performance
 - Message Passing Efficiency
 - Algorithm
 - System Performance

Performance Measurement

Timers

- Automatic profiling or manual timers
- CPU Time
 - Ignore system time
 - Message waiting time
- Wall-Clock Time
 - `sleep 5`
 - CPU Time = 0
 - Wall-Clock Time = 5

Performance Measurement

Shell Time Command

- Limited: Times entire application

```
time sleep 5
0.000u 0.000s 0:05.02 0.0% 0+0k 0+0io 163pf+0w
```

```
time fc
2.840u 0.010s 0:02.86 99.6% 0+0k 0+0io 172pf+0w
```

User time

System time

Wall Clock time

U+S/MC

Size+RSS

I/O access

Page Faults
+Time Swapped Out

Performance Measurement

Manual timers in code

- Insert at start and end of section to be timed
 - Manually track all logic
 - Might have to instrument entire application
- C
 - `gettimeofday(&t, NULL)`
- Fortran
 - `cpu_time(t1)`
 - `system_clock(t1, count_rate, count_max)`
 - `etime(t(:)) , dtime(t(:))`
- MPI
 - `t1 = MPI_Wtime`

Performance Measurement

Gprof

- Automatically tracks times in routines
- Call trees
- Usage
 - Compile entire code with “-pg” INTEL -p, -qp
 - Run code like normal
 - Produces gmon.out
 - gprof gmon.* <exe> > myProfile.out
- <http://www.gnu.org/software/binutils/manual/gprof-2.9.1/>

Performance Measurement

Parsing Gprof Output

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
60.59	13.65	13.65	1	13.65	13.65	block_
30.58	20.54	6.89	1	6.89	6.89	unroll_
3.24	21.27	0.73	1	0.73	0.73	strength_
3.02	21.95	0.68	1	0.68	0.68	pipe_
2.40	22.49	0.54				sin.J
0.18	22.53	0.04				sin
0.00	22.53	0.00	1	0.00	21.95	MAIN__

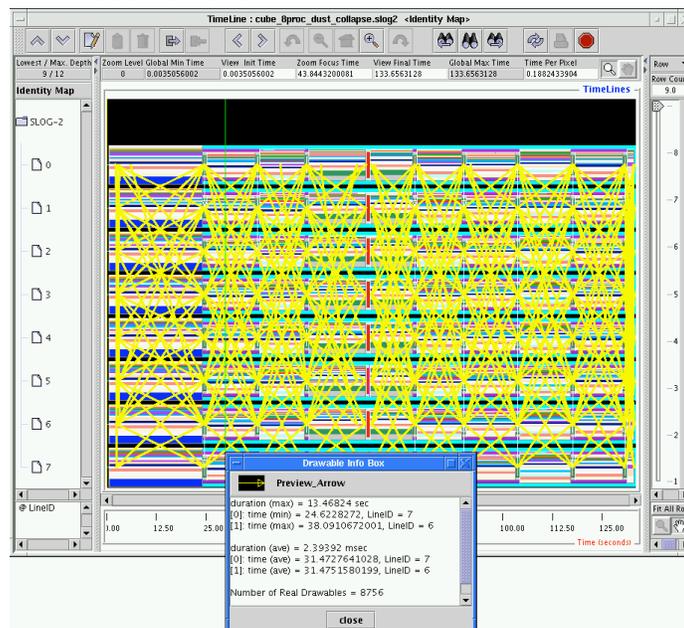
Performance Measurement

Jumpshot

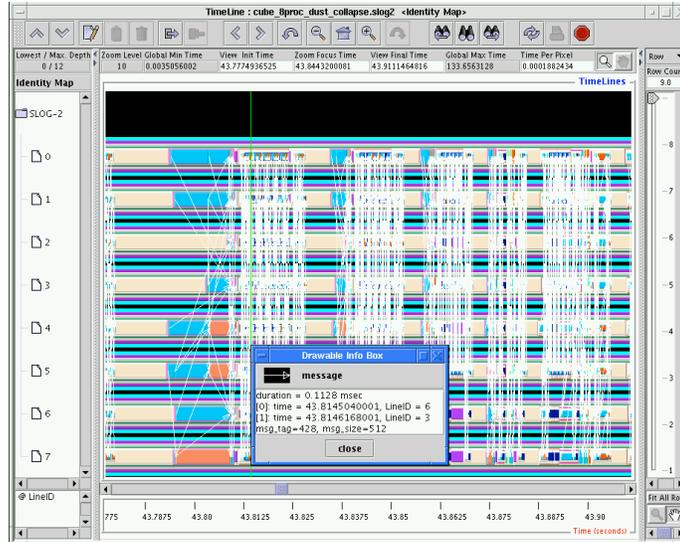
- Automatically profile and visualize MPI
 - Times, communication patterns, message sizes
 - Manually insert profile tags
- Usage
 - Relink MPI app with jumpshot `-lmpilog`
- Run with lots of available space
 - Will generate large clog file
 - `clog2slog`
- With X display, run jumpshot on slog file

<http://www-unix.mcs.anl.gov/mmpi/mpich/docs/mpeman/mpeman.htm>

Performance Measurement



Performance Measurement



Performance Measurement

FPMPI

- Automatically profile MPI routines
 - Total time in MPI routines called
 - Message information per-MPI routine
 - Average, Min, Max, Histogram
 - Potential 2-D mapping
- Relink code with `'-lfpmpi'` and run
- Output : `fpmpi_profile.txt`

Performance Measurement

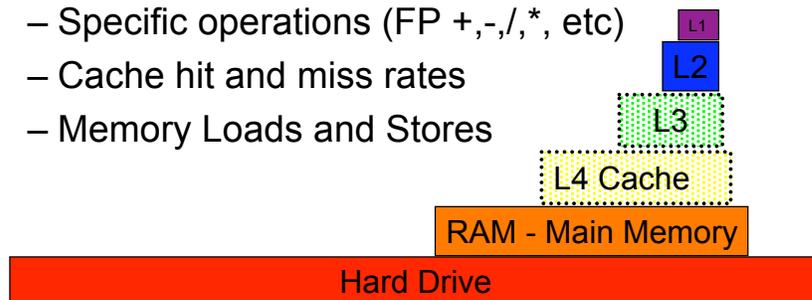
Partial FPMPI Output

```
Date: Fri Feb 4 22:10:53 2005
Processes: 2
Execute time: 861.4
Timing Stats: [seconds] [min/max] [min rank/max rank]
  wall-clock: 861.4 sec 861.310000 / 861.450000 0 / 1
Memory Usage Stats (RSS) [min/max KB]: 20877/20890
Memory Usage Stats (RSS) : 20890 KB
...
Routine Calls ave time min time max time ave len \
MPI_Send 6500000 426.996574 0.000000 636.629670 104000000000 \
MPI_Recv 6500000 426.960924 0.000000 636.605715 104000000000 \
  min len max len Calls by message size \
  0 155200000000 .00000000..112500000000000000 \
  0 155200000000 .00000000..112500000000000000 \
                                     % data by message size
                                     00000000....270000000000000000
                                     00000000....270000000000000000
```

Performance Measurement

Architecture Performance

- Hardware counters
 - Small set of registers that count events
- Events include
 - Specific operations (FP +,-,/,*, etc)
 - Cache hit and miss rates
 - Memory Loads and Stores



Performance Measurement

Performance-API (PAPI)

- Hardware counters for x86
 - Relatively low-level
 - Explicitly instrument your application
- Installation makes it less common
 - Alter linux kernel, easier on windows
 - *NOT* on Jazz
- Example

```
call PAPIF_flops(rtime,ctime,fpins,mfs,ierr)
```

<http://icl.cs.utk.edu/papi/overview/>

Performance Measurement

Valgrind

- Easy tool for memory diagnostics
 - Checks for memory leaks
 - Pseudo-checks cache performance
- No recompiling needed

```
>mpirun -np # valgrind --tool=<tool> <exe>  
    tool=memcheck, cachegrind, ...  
    Get <tool>.out.pid  
>cg_annotate --<pid> cachegrind.out.<pid>
```

<http://valgrind.kde.org/>

Performance Measurement

Cachegrind Output

```
>cg_annotate --12706 cachegrind.out.12706
-----
      Ir  I1mr  I2mr           Dr          D1mr  D2mr           Dw   D1mw   D2mw
-----
3,513,583,467 4,393 2,293 1,737,342,759 103,711,760 25,910 252,245,934 809,356 789,500 PROGRAM TOTALS
-----
      Ir  I1mr  I2mr           Dr          D1mr  D2mr           Dw   D1mw   D2mw  file:function
-----
2,833,939,717  22   11 1,531,111,607 83,500,012      0 251,017,140   6,561   820  ???:unroll_
580,503,243   14    7  180,000,409 20,100,002      1    2,017     206   200  ???:pipe_
94,943,909    15    8   25,250,006   25,155        0  151,517  12,791  314  ???:strength_
-----
-- User-annotated source: cachegrind.out.12706
-----
No information has been collected for cachegrind.out.12706
```

Single CPU Performance

Common CPU Bottlenecks

- Cache problems
 - Little spatial or temporal locality
 - Bad alignment
 - Thrashing
- Unnecessary calculation repetition
- Slow (particularly math) libraries

General Tuning

General Tuning

- Compiler Options
- In-lining
- Loop Parallelism
- Mathematical Libraries

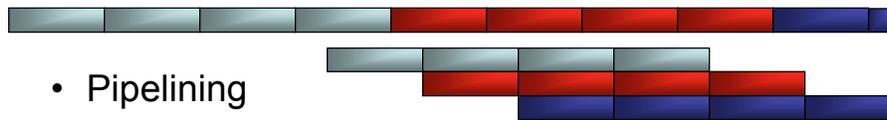
General Tuning : Compiler Options

Compiler Options

- Microprocessor chip options
- All-In-One
- Inter-Procedural Analysis (IPA)
- Loop optimizations
- Numerical optimizations

General Tuning : Compiler Options

Microprocessor Chip Options



- Pipelining
 - Today, mostly in hardware
 - Use the opts if you see them!
- Chip Registers
 - Frequent access, short time
 - `register int i;`
- Pre-fetching data
 - Hide latency
 - Automatically inserted assembly (-O2 -O3)

General Tuning : Compiler Options

All-In-One Opts

- Common `-O n` $n=0-5\dots$, `-fast`, `-qhot`, etc
 - 0 turns all off
 - Read man pages for exact process
- High opts likely to change answers
 - Some to just precision
 - Some make the wrong answer

General Tuning : Compiler Options

IPA : Inter-Procedural Analysis

- Automatic In-lining
 - Replace routine calls with actual code
- Disabling Recursion
- Parameter follow through
- Many other options - read man page

General Tuning : Compiler Options

Loop Optimizations

- Loops have overhead - reduce it
- Compilers will do some
 - Tell compiler how far to go
 - -unroll#, -Munroll
 - Limited by info compiler can determine
- Keep to strides of 1
- Unrolling, Fusion, Linearization

General Tuning : Compiler Options

Loop Unrolling

```
for (i=0; i<160; i=i+1) {  
    a[i] = b[i] + c[i];  
}
```

Unroll by 2

```
for (i=0; i<80; i=i+2;) {  
    a[i] = b[i] + c[i];  
    a[i+1]= b[i+1]+ c[i+1];  
}
```

Unroll by 4

- Compiler
 - Needs indices
 - Dependencies get in way!

```
for (i=0; i<40; i=i+4;) {  
    a[i] = b[i] + c[i];  
    a[i+1]= b[i+1]+ c[i+1];  
    a[i+2]= b[i+2]+ c[i+2];  
    a[i+3]= b[i+3]+ c[i+3];  
}
```

General Tuning : Compiler Options

Loop Fusion and Splitting

- Fusion
 - Combine two loops with same control
 - Rethink code order
- Splitting

```
DO I=1,1000
  (lots of optimizable code
   that need not be in loop)
  ...
  A(I)=B(I)*A(I-1)
END DO
```

General Tuning : Compiler Options

Loop Linearization

- Reduce multi-nested loops
- Loop multi-d arrays in order of memory
- Fortran syntax makes it easy
 - $A(:) = B(:) * C(:)$

```
C: (depends on allocating)
for (i=0; i<imax; i++)
  for (j=0; j<jmax; j++)
    for(k=0; k<kmax; k++)
      A[i][j][k] = 0.0;
```

```
FORTRAN:
do k = 1,kmax
  do j = 1,jmax
    do i = 1, kmax
      do stuff A[i][j][k]
```

General Tuning : Compiler Options

Numerical Optimizations

- MADD Instruction (Itanium)
 - Multiply and add at one time
 - Not IEEE, but, still accurate
- SIMD/Small vector Instructions
- Restricted Pointers (C, C++, etc)
 - “-noalias” and/or “-restricted”
 - Gives pointer info to compiler to permit optimizations
- Precision
 - Operations optimized for word size
 - 32 or 64 bit word size
 - Promote to 128 could hurt by 50%

General Tuning : In-Lining

In-Lining

- Have routines and performance too
- Automatically replace subroutine calls with text of routine
 - Reduce large overhead of calls
 - Permit optimizations
- Implementation
 - IPA does some
 - Direct (`inline int function someFunc(...)`)
- Be aware of code bloat

General Tuning : Loop Parallelization

Loop Parallelization

- Evaluate iteration independence
- Split loop over multiple processes
 - Loop of 1..100 -> 1..50, 51..100
- Be aware
 - Reproducing serial code
 - Updating shared data

General Tuning : Loop Parallelization

Automatic Loop Parallelization

- Insert OpenMP directives with enough information

<pre>subroutine mySub(a,b,c) do i=1,10 read(a(i),b(i),c(i)) enddo enddo</pre>	<pre>SUBROUTINE mySub(a,b,c) INTEGER i DO i=1,10 read(a(i),b(i),c(i)) ENDDO ENDDO</pre>
<p>Cannot know what xfunc does</p>	<p>Indirect reference to data</p>
<pre>subroutine mySub(a,b,c) read(a(i),b(i),c(i)) call xfunc enddo enddo</pre>	<pre>subroutine mySub(a,b,c) INTEGER i integer idx(10) do i=1,10, \ sha(idx(i))=a(idx(i))+c(i) ENDDO</pre>
<pre>subroutine mySub(a,b,c) do i=1,10 a(i)=xfunc(b(i),c(i)) enddo enddo</pre>	<pre>SUBROUTINE mySub(a,b,c) DO i=1,10 a(i)=b(i)+c(i) ENDDO</pre>

General Tuning : Mathemaical Libraries

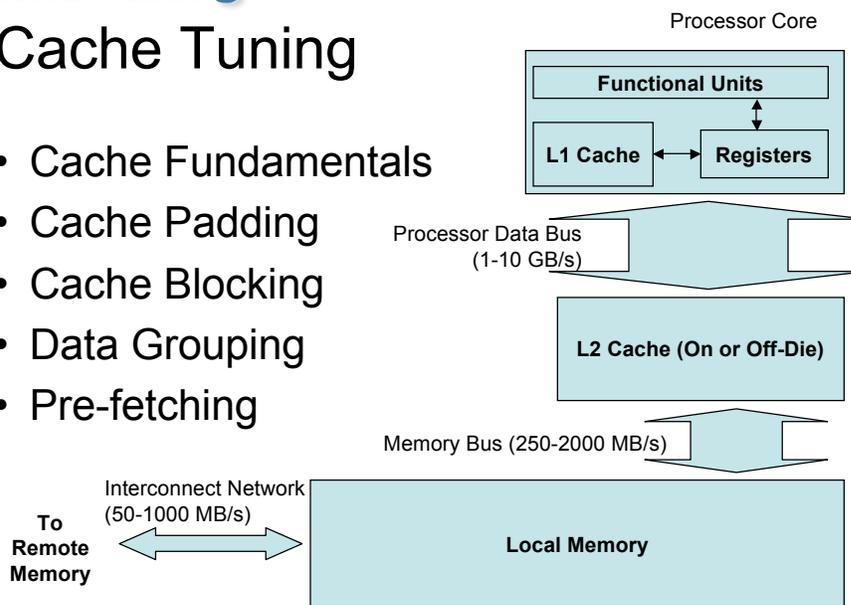
Mathematical Libraries

- Use what exists and had already been optimized for the system
 - “-noieee”
 - Calculator functions
 - BLAS, LAPACK, ScaLAPACK, etc
- Check for faster implementations!

Cache Tuning

Cache Tuning

- Cache Fundamentals
- Cache Padding
- Cache Blocking
- Data Grouping
- Pre-fetching



Cache Fundamentals

- Cache Hit
 - Reference data already in low level cache
- Cache Miss
 - Reference data not in low level cache
- Spatial and Temporal Locality
 - Reading in line or nearby location
 - Reusing data once loaded

Cache Designs

- Direct Mapped Cache
 - Each memory address is mapped physically mapped to a cache line
 - Memory locations share cache locations
- Fully Associative Cache
 - Memory address can have any cache line
 - Complex searching increases latency
- N-way Set Associative Cache
 - Memory address can be mapped to any N cache lines

Cache Tuning : Cache Fundamentals

Data Memory Alignment

- Data cache line makes up cache
 - Normally 4-8, 8-16 bit words
- Store data on word boundaries
 - Compiler flags, directives
- Data fetched into cache is fetched as a line, not just one word
 - A(1) needed, A(2) A(3) A(4) also come
- Many numerical opts need alignment

Cache Tuning

Cache Points Touched

- Loop Linearization - stride one
 - Make use of cache line
 - Faster use of memory
- Loop parallelization/splitting
 - Better fills cache, no overflowing
- Pre-fetching compiler flags
 - Hide the latencies

Cache Tuning : Cache Padding

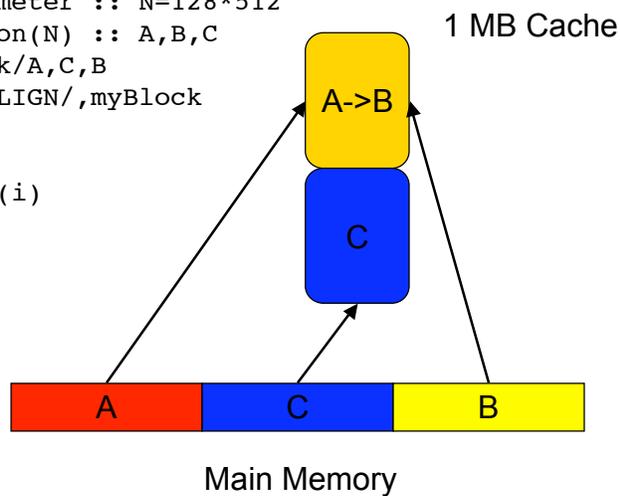
Cache Thrashing

- Two or more data items that are frequently needed by the program both map to the same cache address
- Loop blocking deals with most problems
- Slight offsets in data structures to prevent cache conflict

Cache Tuning : Cache Padding

Cache Thrashing

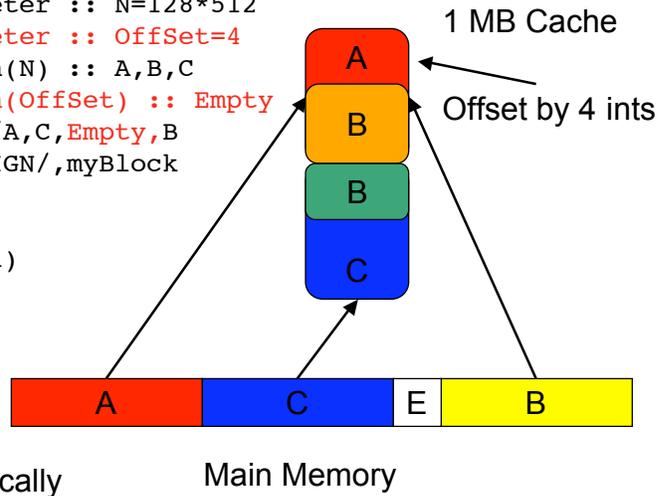
```
integer, parameter :: N=128*512
real, dimension(N) :: A,B,C
common/myBlock/A,C,B
!DIR$ CACHE_ALIGN/,myBlock
...
do i=1,N
  C(i)=A(i)+B(i)
enddo
```



Cache Tuning : Cache Padding

Cache Padding

```
integer, parameter :: N=128*512
integer, parameter :: OffSet=4
real, dimension(N) :: A,B,C
real, dimension(OffSet) :: Empty
common/myBlock/A,C,Empty,B
!DIR$ CACHE_ALIGN/,myBlock
...
do i=1,N
  C(i)=A(i)+B(i)
enddo
```



Cache Tuning : Cache Blocking

Cache Blocking

- Used when referencing data too large to fit in cache
- Spatial reuse
 - Exploit using cache lines
- Temporal reuse
 - Do everything you can to reuse data

Cache Tuning : Cache Blocking

Cache Blocking

Spatial Reuse

```
do j=jmin,jmax
  do i=imin,imax
    p(i,j)=m(i,j)*c*v(j,i)
  enddo
enddo
```



```
p(1,1)=m(1,1)*v(1,1)
p(2,1)=m(2,1)*v(1,2)
p(3,1)=m(3,1)*v(1,3)
p(4,1)=m(4,1)*v(1,4)
```

Temporal Reuse

```
do j=jmin,jmax
  do i=imin,imax
    p(i,j)=m(i,j)*v(j)
  enddo
enddo
```



v(j) is loaded and reused for $\text{imax}-\text{imin}+1$ times.

Cache Tuning : Data Grouping

Data Grouping

- Data structures that encourage locality

Indirect referencing affects performance

```
do i=imin,imax
  j=index(I)
  x=x+sqrt(x(j)^2+y(j)^2+z(j)^2)
enddo
```

Remove indirect reference

```
do i=imin,imax
  x=x+sqrt(r(1,j)^2+r(2,j)^2+r(3,j)^2)
enddo
```

Cache Tuning : Pre-fetching

Pre-fetching

- Compiler hides memory access latency by overlapping memory access and computation
- Best results with minimal use of:
 - Global variables
 - Pointers, type casting
 - Complex flow

MPI Performance Topics

MPI

Amdahl's Law

$$speedup = \frac{1}{\frac{P}{N} + S}$$

N : Number of processes
S: 1-P, fraction of code that is serial

- Potential code speedup is a function of the code that can be parallelized (P)
- Overhead
 - Serial code, latency

MPI

MPI Routine Types

- Blocking Point-to-Point
- Non-Blocking Point-to-Point
- Persistent Point-to-Point
- Completion/Testing Point-to-Point
- Collective Communication

MPI

Quick MPI Review

- Synchronous
 - Completion of send requires initiation of receive
- Non-Blocking
 - Operation does not wait for completion
- Ready
 - Correct send requires matching receive
- Asynchronous
 - Communication and computation happen simultaneously. Varies by MPI implementation.

MPI

Common Performance Factors

- Platform/Hardware
 - CPU, memory subsystem, OS
 - Network adaptors
- Network
 - Hardware
 - Protocols
 - Tuning/Configuration
 - Contention
- Applications
 - Algorithms
 - Comm:Comp
 - Load Balance
 - IO
 - Memory
 - Everything
- MPI Implementation
 - Buffering
 - Protocols(eager, rend..)
 - S-R sync
 - Routine internals

MPI:Buffering

Message Buffering

- The storage of the message between the send and the posting of matching receive
- Implementations differ
 - Send buffers with Sender
 - Send buffers with Receiver
 - No Buffering
- Can create user buffer space

MPI:Buffering

Message Buffering Pros/Cons

- Improve Performance
 - Asynchronous communication
- Compute while communicating
- Less robust
 - Buffer is finite
 - Buffering often well hidden
 - Less portable
- A correct MPI app does not rely on buffer space
 - Unsafe

MPI:Protocols

Message Passing Protocols

- Internal methods and policies implemented by MPI
 - Eager - Assumed buffering by the Receiver
 - Asynchronous protocol that allows send ops to complete w/o acknowledgement of receive
 - Rendezvous
 - Synchronous protocol requiring acknowledgement of matching receive
- Note
 - Implementations differ, not in standard
 - Imp. might use combinations
 - Work with buffering

MPI:Protocols

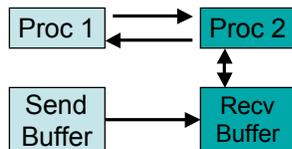
Eager Protocol

- | Pro | Con |
|---|--|
| <ul style="list-style-type: none">• Reduce synchronization delays• Simplifies programming (MPI_Send) | <ul style="list-style-type: none">• Not scalable• Sucks up memory• Bogs down CPU of receiver |

MPI: Protocols

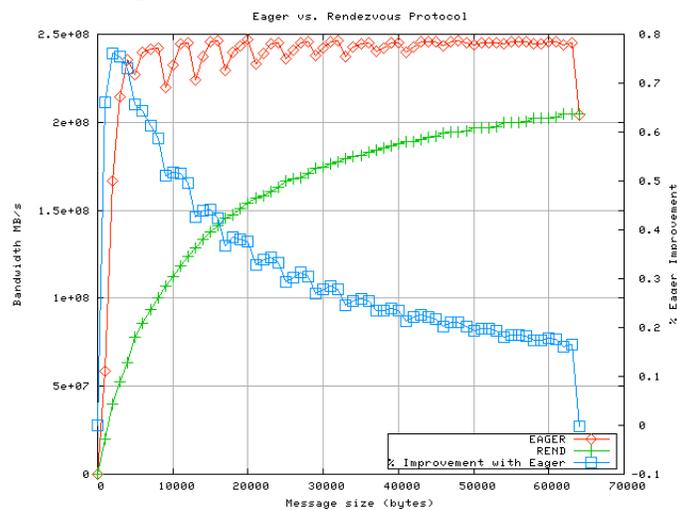
Rendezvous

- No assumptions about the receiver
- When eager protocol exceeded
- Handshaking
- Pros
 - Scalable
 - Robust
 - Possibly avoid copy
- Cons
 - Sync delays
 - Programming complexity



MPI: Protocols

Eager vs. Rendezvous



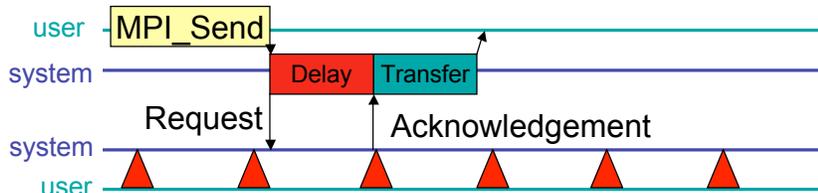
MPI:Synchronization

Synchronization : Polling vs. Interrupt

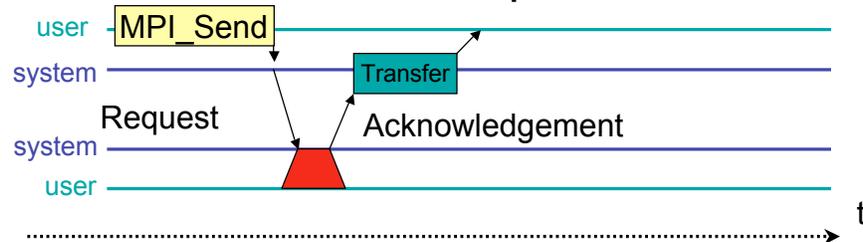
- Use to control cooperation of send and receive tasks
- Implementation dependant
 - How does receive discover the send?
 - How often does receive check for send?
 - How does a process give CPU time to a send?

MPI:Synchronization

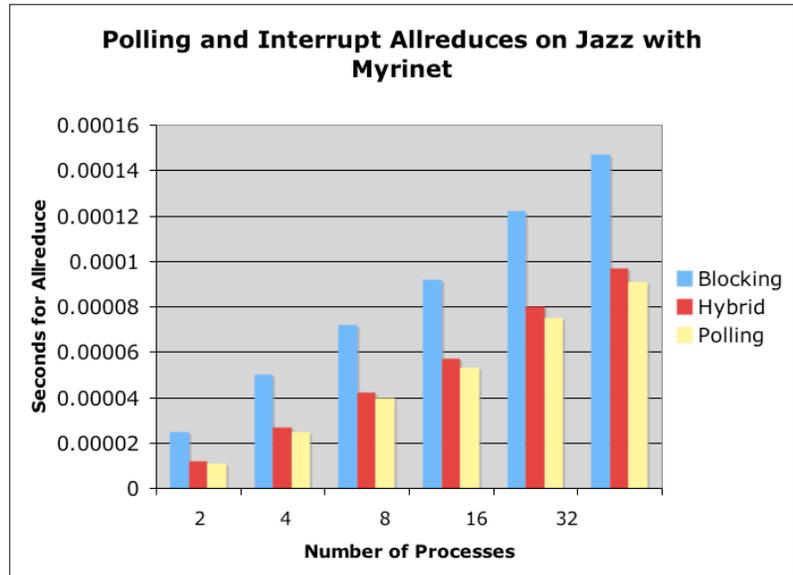
Polling



Interrupt



MPI:Synchronization



MPI:Synchronization

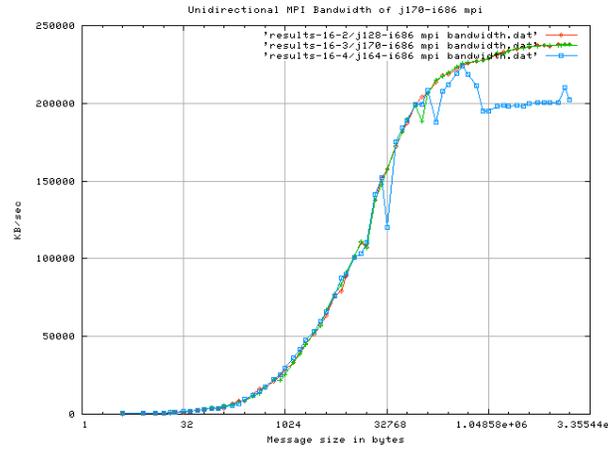
When To Use Interrupt

- Non-Blocking
- Non-Synchronized send-receive pairs
- When waits are not issued immediately after the non-blocking send and receive operations
- Multiple processes per processor

MPI: Message Size

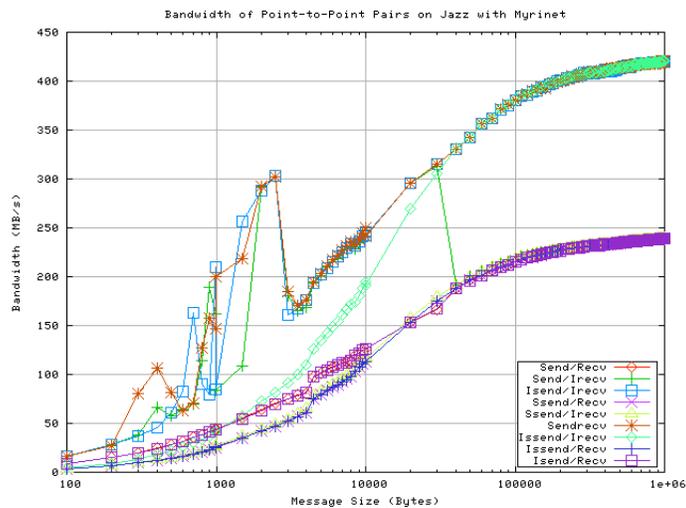
Message Size

- Size of message impacts bandwidth



MPI: Point-to-Point

Point-to-Point Communication



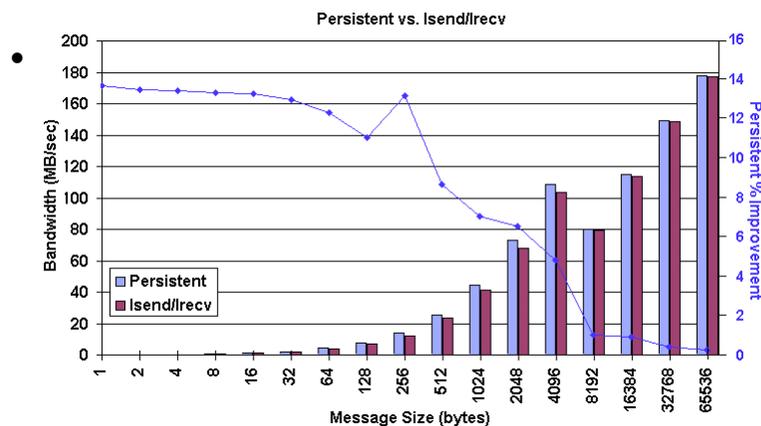
MPI:Persistent

Persistent Communications

- Minimize overhead for point-to-point communications with same arguments
- Persistent communications are non-blocking
- Process
 - Create a buffer
 - Start transmission
 - Wait/Test
 - Clean up

MPI:Persistent

Improvement



MPI:Collective

Collective Communications

- Synchronization is *critical*
- Algorithms are vendor specific
- Might need to explore different hand algorithms with available routines
- MPI-1 specifies blocking collective communication
- MPI-2 defines corresponding non-blocking

MPI:Derived Types

Derived Datatypes

- Construct message
 - noncontiguous memory
 - different datatypes
- Eliminate small message overhead
- Vectors and Hyper-Vectors for non-contiguous types
 - Can hurt performance
 - Memory access and copies

MPI:Derived Types

Derived Datatypes

```
typedef struct {
    float  f1,f2,f3,f4;
    int    i1,i2;
}         f4i2;
f4i2     rbuff, sbuff;
```

MPI_Type_struct
One Send/Recv

2531 KB/s

Individual
Send/Recv Pairs

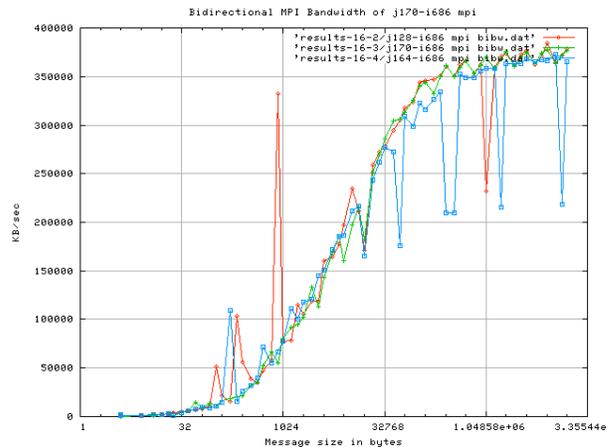
1335 KB/s

Bandwidth for Non-Contiguous Types (MB/s)			
Method	IBM (PPC)	SP2 (P3)	Jazz-Myrinet
MPI_Type_vector	12.1	38.6	102.5
MPI_Type_struct	9.6	38.6	27.7
User pack/unpack	20.2	24.7	103.4
Individual S/R	0.3	0.4	0.8

MPI:Contention

Network Contention

- Saturate the available network
- Not much you can do but be aware



Appendix

Functional Units of Chip

Proc Type	L/S Unit	Integer Unit	FP Unit	Other Units *	OfO?	Speculative
PII+III	2	2	1	1	Y	Y
P4+Xeon	2	3	1	2	Y	Y
Athlon	3	3	3	0**	Y	Y
G4	2	2	1	1	Y	Y
G5	2	2	2	6	Y	Y
Itanium	2	2	2	3	Y	Y
Itanium II	2	4	2	3	Y	Y

Cache Tuning : Cache Padding

Cache Padding

